Lambdascope
Another optimal implementation of the lambda-calculus

Vincent van Oostrom Kees-Jan van de Looij
Marijn Zwitserlood
Universiteit Utrecht
Department of Philosophy
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
{Vincent.vanOostrom,KeesJan.vandeLooij,Marijn.Zwitserlood}@phil.uu.nl

Abstract

An optimal implementation of the $\lambda\beta$-calculus into interaction nets, featuring

1. *only a single type of scope node,*
2. *a completely reduction based read-back, and*
3. *only three reduction rule schemes.*

1. Introduction

We present an optimal implementation of β-reduction on λ-terms. For any λ-term [3] (Section 2), translating it by $\Box$ to an interaction net [9] (Section 3), then performing a number of interaction steps (Section 4), and finally unwinding the resulting interaction net by $\triangle$ to a tree-like net isomorphic to a λ-term again (Section 5), yields a λ-term which is reachable by a number of β-steps (Section 2) from the initial term.

To show correctness we first introduce the set of λ-nets which contains all the nets reachable from translated λ-terms, and for which a stack-based read-back map $\blacktriangle$ from λ-nets to λ-terms can be defined [10, 7] (Section 6). Using the read-back we show that a Beta-step from a λ-net is projected by $\triangle$ onto a multi-Beta-step, performing a number of Beta-steps simultaneously, from the unwound net [2] (Section 7). Correctness follows since by its tree-like form, a multi-Beta-step on a tree-like net, followed by unwinding corresponds to a β-development [3] on its λ-term.

To show optimality [11] it suffices to note that we implement the same abstract algorithm [2] as extant optimal implementations in interaction nets [10, 7, 2]. Although optimality was the original motivation for our studies, we wil not high-light it here. Instead, we focus on presenting the calculus itself.

2. λ-calculus

In order to give a rational reconstruction of our optimal implementation of β-reduction in the λ-calculus [3], we first present a factorisation of β-reduction for the namefree λ-calculus [5] into argument replication and scope extrusion.[1] We employ the following as a running example.

Example 1 *The application $\underline{2}\,\underline{2}$ of the (Church) numeral $\underline{2} = \lambda x.\lambda y.x(xy)$ to itself, reduces to $\underline{4}$ in five steps*

$$\begin{array}{lll} \underline{2}\,\underline{2} & \to_\beta & \lambda y.\underline{2}(\underline{2}y) \\ & \to_\beta & \lambda y.\underline{2}\lambda x.y(yx) \\ & \to_\beta & \lambda y.\lambda z.(\lambda x.y(yx))((\lambda x.y(yx))z) \\ & \to_\beta & \lambda y.\lambda z.(\lambda x.y(yx))(y(yz)) \\ & \to_\beta & \lambda y.\lambda z.y(y(y(yz))) \end{array}$$

(Application of Church numerals is exponentiation.)

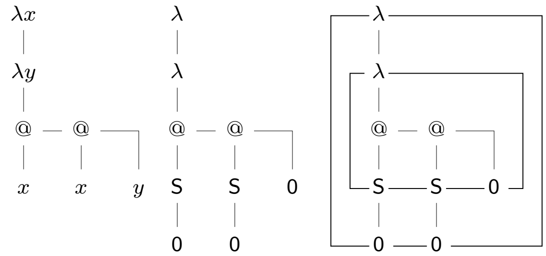
Instead of λ-terms, we implement nameless λ-terms [5].

Example 2 $\underline{2}\,\underline{2}$ *with* $\underline{2} = \lambda\lambda(\mathsf{S0})((\mathsf{S0})\mathsf{0})$ *reduces to* $\underline{4}$

$$\begin{array}{lll} \underline{2}\,\underline{2} & \to_\beta & \lambda\underline{2}(\underline{2}\mathsf{0}) \\ & \to_\beta & \lambda\underline{2}\lambda(\mathsf{S0})((\mathsf{S0})\mathsf{0}) \\ & \to_\beta & \lambda\lambda(\lambda(\mathsf{SS0})((\mathsf{SS0})\mathsf{0}))((\lambda(\mathsf{SS0})((\mathsf{SS0})\mathsf{0}))\mathsf{0}) \\ & \to_\beta & \lambda y\lambda z(\lambda x(\mathsf{SS0})((\mathsf{SS0})\mathsf{0}))((\mathsf{S0})((\mathsf{S0})\mathsf{0})) \\ & \to_\beta & \lambda\lambda(\mathsf{S0})((\mathsf{S0})((\mathsf{S0})((\mathsf{S0})\mathsf{0}))) \end{array}$$

[1] To a large extent our presentation reflects the way our implementation was developed. People not interested in that can fast forward to the next section, after taking note of the inference system for generalised λ-terms.

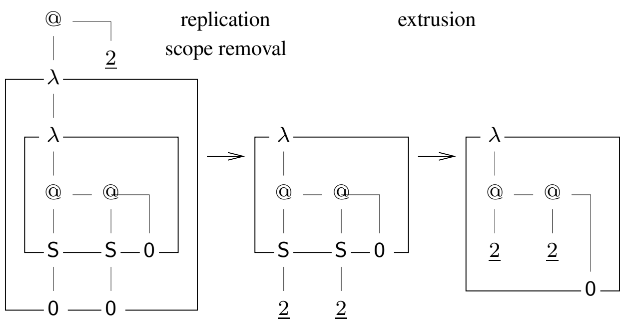
The unary notation for De Bruijn-indices employed here serves the interpretation of successors as end-of-scope operators as introduced in [8]. This interpretation is illustrated for $\underline{2}$ by



displaying from left to right, the syntax tree of the λ-term $\underline{2}$, the syntax tree of the nameless λ-term $\underline{2}$, and that tree again with scopes explicitly indicated by boxes. As illustrated by the figure, and as observed by [4], nameless λ-terms have a context free tree structure: every successor and zero of a closed λ-term *matches* a unique λ; the box is a way to represent this matching explicitly. In other words, the usual notion of binding for named λ-terms is seen to correspond to the notion of matching for namefree λ-terms.

The matching structure can be employed to implement β-reduction as follows. An occurrence of $\mathsf{0}$ in t matching the λ of a β-redex $(\lambda t)s$ has the operational meaning: put the argument s. Dually, an occurrence of S in t matching that λ has the operational meaning: throw the argument s away (as any subterm of this term will be out of the scope of the λ, so to speak). The nameless pendant of the idea that the binding structure is preserved by β-reduction on named λ-terms, is then that matching is preserved in case of nameless λ-terms. As a consequence, β-reducing $(\lambda t)s$ does not only involve replicating the argument s an appropriate number of times, but also managing matching. In the literature on explicit substitutions starting from [1], one varies on both aspects. The starting observation for the variation presented here is that the first β-step of Example 2 can be decomposed as

$$\underline{2}\,\underline{2} \to_\beta \lambda(\mathsf{S}\underline{2})((\mathsf{S}\underline{2})\mathsf{0}) \to_\beta \lambda\underline{2}(\underline{2}\mathsf{0})$$



That is, β-reduction consists of first replicating the argument $\underline{2}$ putting the latter at all $\mathsf{0}$s in the body matching the λ, and then removing the @ and λ of the redex as well as the scope of λ, that is, *all* S*s matching with* λ (in this case none) are elided. (Not removing them would disturb matching and possibly cause dangling 'pointers' or worse.) This results in the generalised λ-term $\lambda\mathsf{S}\underline{2}((\mathsf{S}\underline{2})\mathsf{0})$. Here a generalised λ-term, as introduced in [4], is a λ-term where successors are generalised to be applicable to any (generalised) term instead of just to De Bruijn indices. Finally, to turn the generalised λ-term into the ordinary λ-term $\lambda\underline{2}(\underline{2}\mathsf{0})$ again, successors need to be pushed to the leafs in a matching-preserving way, a process which we call scope extrusion [8].

Formally, the grammar for the set $G\Lambda$ of generalised λ-terms is

$$t \in G\Lambda \quad ::= \quad \mathsf{0} \mid \mathsf{S}t \mid \lambda t \mid tt$$

We employ t, s, u, ... to range over generalised λ-terms and i, j, k, ... to range over its subset of (De Bruijn) indices, i.e. repeated applications of S to $\mathsf{0}$. Ordinary nameless λ-terms are obtained by requiring successors to occur as part of indices. β-reduction on ordinary (nameless) λ-terms factorises then as follows. First replication and removal are performed according to

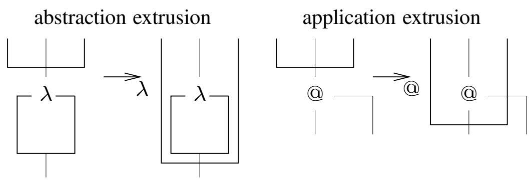
$$(\lambda t)s \to t[s]^{\mathsf{0}}$$

with 'substitution' $t[s]^i$ of s in t at depth i defined by

$$\begin{array}{rcl}
\mathsf{0}[s]^{\mathsf{0}} & = & s \\
\mathsf{0}[s]^{\mathsf{S}i} & = & \mathsf{0} \\
(\mathsf{S}t)[s]^{\mathsf{0}} & = & t \\
(\mathsf{S}t)[s]^{\mathsf{S}i} & = & \mathsf{S}t[s]^i \\
(\lambda t)[s]^i & = & \lambda t[s]^{\mathsf{S}i} \\
(t_1 t_2)[s]^i & = & t_1[s]^i t_2[s]^i
\end{array}$$

Next scopes are extruded by reducing to normal form with respect to the *scope extrusion* rules

$$\begin{array}{rcl}
\mathsf{S}\lambda t & \to_\lambda & \lambda t^{\mathsf{S0}} \\
\mathsf{S}(t_1 t_2) & \to_@ & \mathsf{S}t_1\mathsf{S}t_2
\end{array}$$



where, for index i, *minimal lifting* t^i is defined by:

$$\begin{array}{rcl}
t^{\mathsf{0}} & = & \mathsf{S}t \\
\mathsf{0}^{\mathsf{S}i} & = & \mathsf{0} \\
(\mathsf{S}t)^{\mathsf{S}i} & = & \mathsf{S}t^i \\
(\lambda t)^{\mathsf{S}i} & = & \lambda t^{\mathsf{SS}i} \\
(t_1 t_2)^{\mathsf{S}i} & = & t_1^{\mathsf{S}i} t_2^{\mathsf{S}i}
\end{array}$$

Note that using the extrusion rules, the first step of Example 2, indeed factorises in the way which was displayed above. In particular, note that $\mathsf{S}\underline{2} \rightarrow_\lambda \underline{2}$ holds since $\underline{2}$ is closed. Here a generalised λ-term t is closed if $0 \vdash t$ in the following inference system (cf. [6]):

$$\frac{\mathsf{S}i \vdash 0}{}\,0 \qquad \frac{\mathsf{S}i \vdash \mathsf{S}t}{i \vdash t}\,\mathsf{S} \qquad \frac{i \vdash \lambda t}{\mathsf{S}i \vdash t}\,\lambda \qquad \frac{i \vdash t_1 t_2}{i \vdash t_1 \quad i \vdash t_2}\,@$$

The intuitive reading of the index i in a judgment $i \vdash t$, read: term t is well-formed under index i, is as the (number of) variables bound by λs above this subterm.

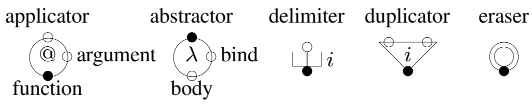
Example 3 *That $\underline{2}$ is indeed closed is witnessed by*

$$\dfrac{0 \vdash \lambda\lambda(\mathsf{S}0)((\mathsf{S}0)0)}{\dfrac{\mathsf{S}0 \vdash \lambda(\mathsf{S}0)((\mathsf{S}0)0)}{\dfrac{\mathsf{SS}0 \vdash (\mathsf{S}0)((\mathsf{S}0)0)}{\dfrac{\mathsf{SS}0 \vdash \mathsf{S}0}{\dfrac{\mathsf{S}0 \vdash 0}{}\,0}\,\mathsf{S} \quad \dfrac{\mathsf{SS}0 \vdash (\mathsf{S}0)0}{\dfrac{\mathsf{SS}0 \vdash \mathsf{S}0}{\dfrac{\mathsf{S}0 \vdash 0}{}\,0}\,\mathsf{S} \quad \dfrac{\mathsf{SS}0 \vdash 0}{}\,0}\,@}\,@}\,\lambda}\,\lambda$$

As usual, any λ-term can be closed (made well-formed under 0), by putting enough abstractions. Hence it is no restriction to prove our results for closed terms only and we will do so. Moreover, we will abbreviate indices by natural numbers in sans-serif e.g. $\mathsf{SSS}0$ is abbreviated to $\mathsf{3}$.

3. From terms to nets

We present our translation of the namefree λ-terms to a class of graphs known as interaction nets [9]. The signature of an interaction nets consists of symbols each having a number of ports among which a designated *principal* port. The interaction net signature we employ is

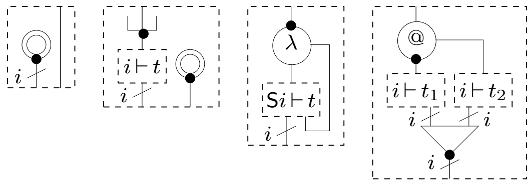


where ∘s indicate ports and •s indicate principal ports, i.e. ports along which a symbol may interact (see the rules below). Here i ranges over arbitrary indices, making the signature infinite.

Apart from @ and λ which will have the meaning one expects, the signature has symbols for explicitly representing the different operations of the factorisation of β-reduction, as presented in the previous section. In particular, the duplicator ∇_i (share, fan) and the eraser $\circledcirc$ (garbage) will together serve to represent replication, as usual in graph implementations of first-order rewriting. The delimiter $\sqcup_i$ represents the higher-order aspect of scope. By default, when we do not write the index i for a duplicator or delimiter, it is assumed to be 0.

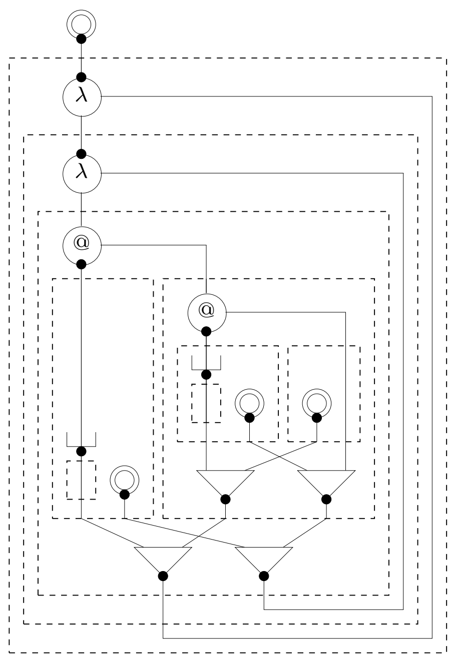
Interaction nets are graphs the nodes of which are labelled by symbols of the signature, and the edges of which connect to the ports of the (symbols of the) nodes. To every port at most one edge may be connected. If no edge is connected to a port, then the port is called free. A net is closed if it does not have free ports.

The function $\square : \Lambda \rightarrow \mathrm{IN}$ mapping closed terms to closed interaction nets is defined in two phases. First, a well-formed term $i \vdash t$ is mapped to a net having $i+1$ free ports, which is defined by induction and cases (0, S, λ, and @) on the definition of well-formedness as:

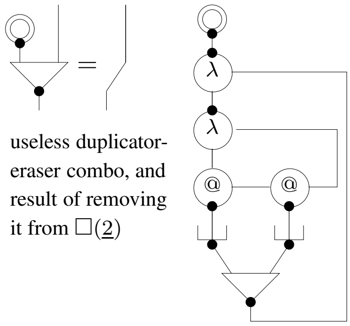


After that a $\circledcirc$-node (the *root*) is connected to the free port. Here a number i next to a slashed edge represents that in fact the edge is a 'bus' consisting of i edges.

Example 4 *The translation $\square(\underline{2})$ of $\underline{2}$ is recursively obtained from the inference displayed in Example 3*



Since the translation is uniform, it is on the one hand easy to prove properties about, but on the other hand very inefficient: it generates many duplicator-eraser combinations whose net-effect will be (see the reduction rules below) the same as that of an edge. Removing these yields the net $\overline{2}$

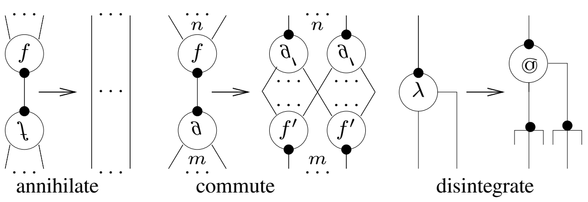


useless duplicator-eraser combo, and result of removing it from $\square(\underline{2})$

where the north port of the second application has been rotated to the west, in order to highlight the correspondence between the interaction net representation of $\underline{2}$ and its syntax tree, as displayed above.

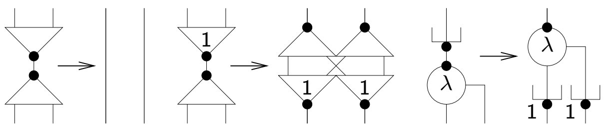
4. Interaction net reduction

The intuitive meaning of the symbols in our interaction net signature, as presented above, is operationalised by just two rule schemes, for f,g arbitrary but distinct, and a rule

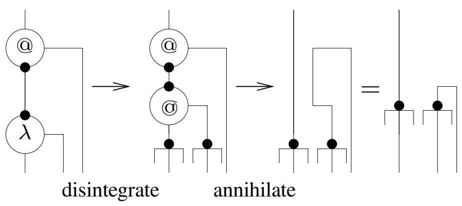


where f' and g' are either identical to or updates of the symbols f and g, respectively. An update is an increment of the index i (if any) of either symbol, which takes place iff the other symbol is either λ or $\sqcup_j$, with $i \geq j$. Instances of the two schemes are called $\mathbf{x}$-rules.

Example 5 *Annihilate, commute, and commute with update, respectively, are exemplified by the following* $\mathbf{x}$*-rules*



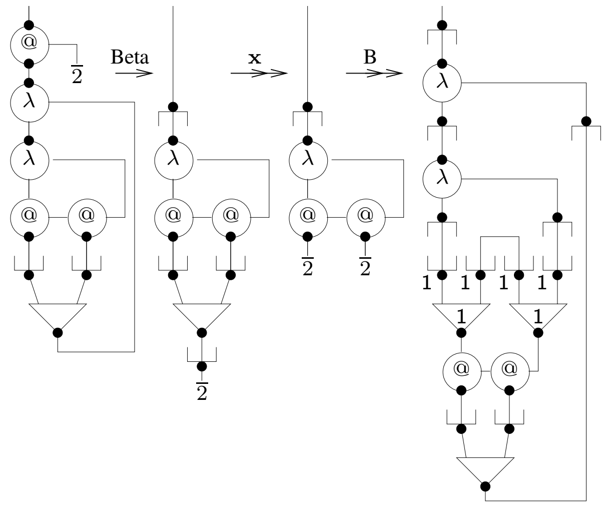
Disintegration is only half of a rule; the rule Beta is defined by post-composing it with an annihilation of its @:



The set B of interaction rules which interest us is defined to be the union of $\mathbf{x}$ and Beta (but not disintegrate).

Note that the effect of operators is indeed as expected: the eraser acts as a garbage collector erasing anything it can interact with; the duplicator acts as a copier duplicating anything it can interact with; the delimiter acts as an extruder putting anything it can interact with into its scope. All act locally in the sense that they affect one node at the time. This restriction, which comes with the interaction net framework, explains our use of *indexed* delimiters: roughly speaking, the way in which the recursive definition of minimal lifting of the previous section is implemented, is to record the superscript there as an index to the delimiter.

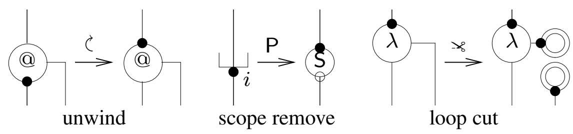
Example 6 *We display only a few nets along the reduction of* $\overline{2}$ *applied to itself, to B-normal form. The final net shown is the normal form which is a representation of* $\underline{4}$. *How to retrieve this term from the net is the topic of the next section.*



5. From nets to terms

The function $\triangle : \mathrm{IN} \to \Lambda$ mapping closed nets to closed terms is defined in three phases, each consisting of normalising w.r.t. an action first and the $\mathbf{x}$-rules next. (Without touching the root-$\circledcirc$.) This yields the syntax tree of a unique (derivation of a) term, which is taken as the result of $\triangle(\nu)$.

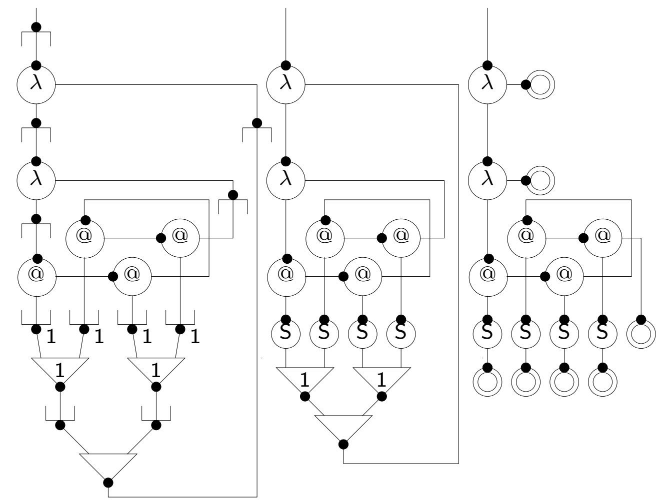
The three phases are named after their actions given by:



Here the S is a new node type, the interaction of which is governed by the $\mathbf{x}$-rules, i.e. S behaves as a non-indexed $\sqcup_i$.

After the unwinding action, both abstractions *and* application have their north port as principal port. This makes that all replication and delimiter nodes are 'pushed toward the leafs/variables' (causing unsharing and extrusion) by the subsequent $\mathbf{x}$-normalisation phase, except for some upward directed delimiters with index 0. The latter are pushed toward the leafs as well in the subsequent scope removal phase, which also puts all replication nodes closer to the leafs than the delimiter nodes. Finally, cutting the loop causes all replication nodes to vanish by having them interact with the eraser, yielding a tree proper.

Example 7 *The* $\mathbf{x}$*-normal forms of the three phases applied to the normal form in Example 6 are, respectively*



From the final tree it is trivial to reconstruct the term $\underline{4}$.

Until now, we have presented a translation of nameless λ-terms in interaction nets, provided some reduction rules on the latter, and some more rules for retrieving a λ-term from nets again. The examples suggest that we have implemented β-reduction. It remains to prove this.

6. Static correctness

In order to prove correctness of our implementation, we provide a series of three, each time more stringent, characterisations of the nets reachable from translated λ-terms, culminating in the set of λ-nets. This set satisfies the following three desirable properties:

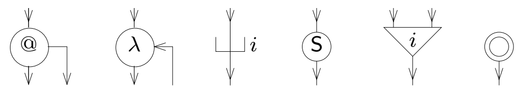
- The translation $\Box$ maps into λ-nets.
- The set of λ-nets is closed under B-reduction.
- The translation $\triangle$ maps λ-nets to λ-terms.

This shows static correctness in the sense that B-reduction can then be viewed as a transformation on λ-terms. In the next section dynamic correctness will be shown in the sense that the transformation is not just any transformation, but that it corresponds to β-reduction.

Unfortunately, we did not succeed yet in proving static correctness directly using our *syntactic* reduction-based function $\triangle$. Instead, we fall back to a *semantic* stack-based read-back function $\blacktriangle$ mapping nets to λ-terms in the spirit of [10, 7], for proving static correctness. We then conclude by noting that this read-back function $\blacktriangle$ coincides with $\triangle$ on the class of λ-nets. The reason why we think this is unfortunate, except for causing a detour, is that the semantic read-back map $\blacktriangle$ is quite complex. For this reason, we only include proof ideas in this and the next section. (However, note that the semantic read-back is only needed in our proof of correctness; the actual implementation does not need it and is extremely simple to implement (just three rule schemes).)

6.1. Directed nets

The interaction nets yielded by the function $\Box$ can be turned into (*rooted*) *directed* nets in DN, by directing the edges in the function $\Box$ as:



Because of the reversely directed east port edge of λ, there must also be reverse versions of the symbols commuting with λ, i.e. of $\sqcup$, S, $\triangledown$ and $\circledcirc$, which are obtained from the above by reversing all arrows for them.

Theorem 8 $\Box$ *maps into DN, which is reduction closed.*

The proof of the first part consists in directing the edges in the translation in Subsection 3 such that they cross the dashed box borders downward.

For a proof of the second part, it suffices to show that for each interaction rule whose left-hand side can be directed, its right-hand side can be directed *with the same directed interface*. This is easy. (Note that although the right-hand side of the disintegrate rule cannot be directed, it combines with @-annihilation to yield the Beta-rule, the right-hand side of which *can* be directed).

From the theorem it follows that the annihilation rules for @ (on its own) and λ are superfluous, as their left-hand sides cannot be directed.

6.2. Tree nets

The read-back function (cf. [7]) $\blacktriangle$ maps tree-nets (to be defined below) to (potentially infinite) trees. It will be defined in three phases, which can be intuitively understood as follows.

Recall that the set of nameless λ-terms is context free. That is, one can define a push-down automaton for recognising nameless λ-terms. Correspondingly, the class of tree nets will consist of nets which are context free in the sense that they are recognised by a 'generalised push-down automaton' (GPDA). This automaton, which will be defined in the first phase, is generalised in that its stack has more structure and allows for more operations than usual PDAs.

In the second phase, we show how the walks of the automaton can be combined to form a tree, which will be the tree read back from the net.

Recall from the first section, that a variable being bound by a λ-abstraction in the named λ-calculus, was rendered in the nameless λ-calculus as an index 0 *matching* with a λ-abstraction. For nets, the corresponding notion is that of a *binding loop* on a λ-node, which will be introduced in the third phase. Just as proving β-reduction of $(\lambda x.M)N$ correct involves showing that substituting N for x in M yields a correct term again, in the case of nets Beta-reduction will cause cutting the binding loop and reconnecting it to the argument and one has to show that this yields a correct walk again. This will be enforced in the third phase, by requiring all binding loops to be transparent in the sense that they do not modify the stack; hence once cut, they can be used to lengthen any path.

After this intuitive explanation, we proceed with the first phase.

6.2.1. The automaton

Any directed net ν is mapped to a directed graph $\mathcal{G}_\nu$ (the 'GPDA') as follows.

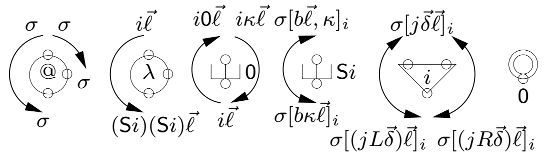
A vertex of $\mathcal{G}_\nu$ is a pair (e,σ) with e an edge of ν and σ a stack, where the grammars for the syntactic categories of, respectively, *blocks*, *levels* and *stacks* are:

$$\begin{array}{rcl} b\in B & ::= & i\vec{\delta} \\ \ell\in L & ::= & b\vec{\ell} \\ \sigma\in S & ::= & i\vec{\ell} \end{array}$$

Here i ranges over indices as before, and δ ranges over the set $\Delta=\{L,R\}$ of *directors*. For a stack $i((j\vec{\delta})\vec{\kappa})\vec{\ell}$, i and j are called its *body* and *bind* indices, respectively.

Intuitively the body index of stack represents how many λs have been visited thusfar, and the stack records all the matching information encountered thusfar, both *qua* sharing as well as *qua* scoping.

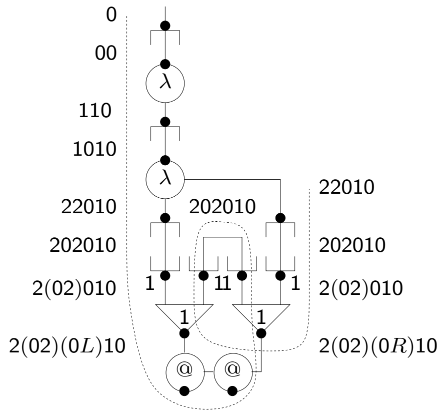
There is an edge from (e_1,σ_1) to (e_2,σ_2) in $\mathcal{G}_\nu$ if the target of e_1 and the source of e_2 are ports of the same node in ν, which are connected according to (the arrows in):



such that the stacks σ_1 and σ_2 satisfy the constraints specified. Here we have employed the context-notation $\sigma[\vec{\ell}]_i$ to denote a stack $\sigma=j\kappa_0,\ldots,\kappa_k$ where κ_i has been replaced by $\vec{\ell}$, implicitly assuming that $i\leq k$.

A *read-back* path of ν is a path in $\mathcal{G}_\nu$ starting at $(\varrho,0)$, where ϱ is the unique edge from the root of ν. A *read-back* stack is a stack occurring as the second component of a vertex along a read-back path.

Example 9 *We show the stacks along a read-back path along the final net in Example 6, where we have omitted the part of the net irrelevant to this path, for clarity.*



6.2.2. Reading back trees

Second, the graph $\mathcal{G}_\nu$ is turned into a rooted tree $\blacktriangle(\nu)$. The vertices of $\blacktriangle(\nu)$ are the vertices (e,σ) of $\mathcal{G}_\nu$ such that

- $e=\varrho$. Then the vertex (e,σ) is the *root*-vertex, or
- the source of e is the body port of a λ. Then e is called a *body* edge and the vertex a λ-*vertex*, or
- the target of e is the bind port of a λ, in which case e is called a *bind* edge. Then the vertex is a $\mathsf{S}^{i-j}(0)$-*vertex*, for i and j the body and bind indices of σ, or
- the target of e is the north port of an @ (an @-*vertex*).

There is an edge in $\blacktriangle(\nu)$ from vertex v to w if there is a path from v to w (as edges!) in $\mathcal{G}_\nu$ through edges not in $\blacktriangle(\nu)$ (as vertices!). Note that paths are deterministic except on @s.

Example 10 *The read-back path displayed in Example 9 witnesses that the rooted tree* $\blacktriangle(\nu)$ *looks like* $\lambda\lambda?(?(?(?0)))$*, where the* $?$*s are unknowns to be determined by the other read-back paths (they will all be* 1*s). Note that the* 0 *is caused by the body and the bind index of the final stack* 22010 *both being* 2*, hence their difference is* 0*.*

6.2.3. Binding loops

Third, the set TN of tree-nets is the subset of DN consisting of nets ν such that any finite ν-walk is a binding loop, where a *ν-walk* is a read-back path of ν which cannot be extended since it either ends in a bind edge or it is infinite, and a walk is said to be a binding loop if it has a cyclic transparent suffix. Here, a path from (e_1, σ_1) to (e_2, σ_2) is

- *cyclic* if the source of the body edge e_1 is the target of the bind edge e_2, and the body index of σ_1 is the bind index of σ_2.
- *transparent* if $\tilde{\sigma_1} \sqsubseteq_i \tilde{\sigma_2}$ with i the body index of σ_1, and where $\tilde{\sigma}$ denotes σ with its body index replaced by $\mathsf{0}$.

Informally, transparency means that σ_1 and σ_2 are identical except that the latter may have been extended with extra (sharing) data on top of the stack (i.e. for the variable introduced by the abstraction).

Example 11 *The path displayed in Example 9 is a binding loop for the suffix starting with the body edge of the second λ encountered:*

- *The suffix ends with a bind edge toward that same λ node, and the respective body indices are both* $\mathsf{2}$.
- *The suffix is transparent as both stacks are identical, hence still identical after replacing the body index by* $\mathsf{0}$.

To formalise a parametric notion of extension ($\sqsubseteq$) it is convenient to employ an alternative representation of stacks, which enables a definition of *extension* of a stack by means of substitution. The alternative representation is obtained by replacing each $\mathsf{0}$-index in any block of a stack with a variable, and by putting a variable to the right of any vector in it, in such a way that variables occur at most once. (This is analogous to the fact that a term-based definition of strings, representing letters by unary function symbols and the tail of the string by a variable, allows for the definition of extension (concatenation) by means of substitution.)

The *higher* than relation on levels is generated by letting the level $b\ell_0, \ldots, \ell_k$ and levels in $\ell_{\mathsf{S}i}$ all be higher than any level in ℓ_i. For a stack $i\ell_0, \ldots, \ell_k$ the situation is reverse: any level in ℓ_i is higher than the levels in $\ell_{\mathsf{S}i}$. Variables and (indices) of blocks are related according to their level. Finally, stack *extension* is then defined as

$$\sigma \sqsubseteq_i^{\vartheta} \tau, \text{ if } \sigma^{\vartheta} = \tau \text{ for some } \vartheta$$

with ϑ a syntactic-category-preserving substitution which is the identity on variables lower than i. Here, we implicitly assume if $i \neq \mathsf{0}$, that i occurs in both σ and τ.

Theorem 12 $\Box$ *maps into TN, which is B-reduction closed.*

If $\nu \rightarrow_{\mathbf{x}} \mu$ one even has $\blacktriangle(\nu) \simeq \blacktriangle(\mu)$. The proof is analogous to the proofs of similar results in the literature e.g. [10, 2]:

- In the case of $\mathbf{x}$-steps one shows that there is a bijective correspondence between the walks before and after the step.
- In the case of a Beta-step one shows that a walk after the step can be obtained by cutting and pasting walks before the step. In order for this to be a proper walk, transparency is crucial.

6.3. λ-nets

The set ΛN of λ-nets is the subset of TN consisting of nets having only finite walks. As a consequence the read-back function yields finite terms, i.e. $\blacktriangle : \Lambda\text{N} \rightarrow \Lambda$.

Theorem 13 $\Box$ *maps into ΛN, which is B-reduction closed.*

6.4 $\triangle \simeq \blacktriangle$

To prove this isomorphism, we show

$$\begin{aligned} \blacktriangle(\nu) &\simeq \blacktriangle(\triangle(\nu)) && (1) \\ \blacktriangle(\mu) &\simeq \mu && (2) \end{aligned}$$

for every λ-net ν, and any tree-like λ-net ν, from which we conclude since $\triangle(\nu)$ is tree-like by construction.

To show (1) it suffices to show that $\blacktriangle$ is invariant for the interactions in each of the three phases of $\triangle$. First we show that $\triangle$ is well-defined, since each of its three phases is.

Proposition 14 $\rightarrow_{\mathbf{x}}$ *is complete (confluent and terminating) on ΛN, also after* ☇, P, ✂, *and it preserves* $\blacktriangle$.

Let us sketch why this proposition holds. Completeness holds since confluence holds for any interaction net by design, and termination follows from read-back paths being finite.

Preservation of $\blacktriangle$ holds for the first phase ☇, since in it only $\mathbf{x}$-rules are applied, and these were already seen to preserve $\blacktriangle$ above.

Since after the first phase a read-back path is a path through the net which is assumed to be in $\mathbf{x}$-normal form, it consists of a number of edges *from* principal ports, followed by a number of edges *to* principal ports. By the assumption that the read-back path does not get stuck, the edges *from* principal ports cannot arise from arrows along $\sqcup_{\mathsf{S}i}$- or ∇_i-nodes, as these require structure which is not present (on the stack). Hence, a read-back path first visits a number of @-, λ- and $\sqcup_0$-nodes *from* their principal port, followed by visits to $\sqcup_i$- and $\triangledown_i$-nodes *to* their principal ports. As a consequence, the net is tree-like in the sense that a read-back path nevers visits a node twice, for this would require passing a node with in-degree greater than 1, hence a $\triangledown_i$-node but then the tail of the path would be a cycle consisting only of edges *to* the principal port, hence either the path gets stuck, if some $\sqcup_i$-node is on the cycle since passing these decrements the (finite) height of the stack, or it is infinite, if only $\triangledown_i$-nodes are on the cycle, both of which are impossible by assumption. Now to show invariance of $\blacktriangle$ holds for the second phase P, note that mapping all indices to 0 preserves read-back. Essentially, this holds since tree-likeness implies that the vertices associated to bind edges have as index i, the difference between the ordinary and the reverse occurrences of $\sqcup$-nodes on their binding loop. Hence letting S have the same semantics as $\sqcup_0$, the result follows since i is easily seen to be invariant under $\rightarrow_{\mathbf{x}}$.

Finally, to show invariance in the third loop-cutting-phase $\rightarrowtail$ and of (2) the invariant above suffices, giving as semantics of *entering* $\circledcirc$-nodes, the semantics of bind edges.

7 Dynamic correctness

From the above, we know that λ-nets are closed under B-reduction. It remains to show that this reduction implements β-reduction and not some other transformation on λ-terms. The proof is along standard lines of reasoning in the area of explicit substitutions [1], showing that unwinding a Beta-step on a λ-net ν unwinds to a Beta-multistep on the unwinding of ν. As for explicit substitution calculi, to show commutation of Beta-steps would involve showing a variant of the substitution lemma, arising from the 'critical pairs' between β-redexes (before $\circlearrowright$) and the $\mathbf{x}$-steps (of $\circlearrowright$). Instead, we show that $\blacktriangle$ commutes with Beta-steps. This is easier (the hard work having been done in Section 6), since $\blacktriangle$ is more semantic than $\triangle$.

Proposition 15 *If $\nu \twoheadrightarrow_{Beta} \mu$ and if the λ-net ν unwinds to ν', then ν' can be unwound further to a λ-net ν'' such that $\nu'' \twoheadrightarrow_{Beta} \mu''$ with $\blacktriangle(\mu) \simeq \blacktriangle(\mu'')$.*

By Section 6, it follows that if $\nu \rightarrow_{\mathrm{Beta}} \mu$, then $\Box(\nu) \twoheadrightarrow_{\mathrm{Beta}} \mu'$, for some net μ' with $\blacktriangle(\mu) \simeq \blacktriangle(\mu')$. Moreover, also by the previous section, applying $\triangle$ to μ' yields a tree-like unwound net isomorphic to $\triangle(\mu)$.

Roughly speaking, an arbitrary B-reduction on nets projects onto a reduction between tree-like unwound nets, such that each of its 'steps' consists of a multi-Beta-steps followed by the three phases of $\triangle$, i.e. by substitution normalisation. Correctness follows, since on tree-like unwound trees this is just the usually procedure for Bourbaki-graphs.

8. Related and further work

Since there is a lot of related and further work, we only briefly touch upon some issues.

8.1. Optimal vs. efficient

Our calculus is an optimal implementation of the λ-calculus in the sense of [11], i.e. all β-redexes having the same Lévy label in a λ-term along a reduction will be represented by the same Beta-redex in the corresponding λ-net. A prototype implementation, which we have dubbed *lambdascope*, shows that out-of-the-box, our calculus performs as well as the *optimized* version of the reference optimal higher-order machine BOHM (see [2]) (hence outperforms the standard implementations of functional programming languages on the same examples as BOHM does).

However, being optimal does not necessarily imply being the most efficient. Indeed, the presented calculus is not optimised, so it would be interesting to try to apply existing optimisation techniques to it.

For instance, extruding a scope over a closed λ-term costs time linear in the size of the term in our implementation, whereas one observes that in such cases it would be safe to simply remove the scope. In order to be able to implement this, one should be able to observe whether a λ-abstraction is closed or not. In the approach of [13] this is possible, explaining in some sense its efficiency. Since at first sight, both the present approach and that of [13] seem to be compatible, it would be interesting to attempt to unify them. More generally, a goal worthwhile pursuing is to try to lift standard first-order optimization techniques to λ-nets.

For another instance, our calculus is based on the untyped λ-calculus whereas most of the time λ-terms will be typable. One would expect a better translation function $\Box$ taking advantage of the type information.

8.2. Disconnected vs. connected scopes

Since in our approach scope delimiter nodes are not directly connected (by an explicit edge or via other scope nodes) to their matching opening λ-abstraction node, there is no way to remove them locally when performing a β-step. However, simply removing the λ-abstraction would leave dangling the closing scope nodes matching it. Our solution was to not remove the scope at all, instead mimicking the scope opening effect of the disappeared λ-abstraction by adjoining explicit scope opening operators, i.e. the $\sqcup_0$-nodes in the disintegrate rule in Section 4.

Another solution would be to *make* the scope nodes matching a λ-abstraction local to it, by explicitly connecting the formers to the latter. Then they can be removed upon performing a β-reduction. This is the approach taken in [13].

8.3. The ideal explicit substitution calculus?

Although we do not show it here, we claim the presented implementation of the λ-calculus has all properties one might desire of an explicit substitution calculus, safe being a *term* calculus.

8.4. Scoping first vs. replication first

Existing work on the optimal implementation of the λ-calculus, e.g. [10, 7, 2], has focused on dealing with explicit local replication, viewing the explicit scope operators (brackets and croissants) only as a necessary evil needed to implement the so-called oracle. Instead, we have followed the opposite route as was suggested in [8] There, an extension of the λ-calculus with an explicit global scope operator called ⅄ was introduced, leaving replication implicit. The ⅄-operator is a generalisation of the generalised successor of [4], which in turn is a generalisation of the successor on De Bruijn indices [5]. The ⅄-operator corresponds to a $\sqcup$-node here. In [15], the global scope operator ⅄ is made local, still leaving replication implicit. Roughly speaking, this amounts to making scope extrusion and in particular the inductive definition of minimal lifting of Section 2 local, leading to the introduction of *indexed* generalised successor (and predecessor) nodes, corresponding to the $\sqcup_i$-nodes (and their inverses) here.

The advantage of following this route is that once explicit (local) scoping is in place, adjoining explicit replication both in its local [14] and global version proceeds analogous [12] to the way replication is made explicit and local in the first-order case.

8.5. Conclusion

We have presented an implementation of the λ-calculus in the spirit of the calculational approach of [5], and which is fully in the traditions of calculi with explicit substitution and of graph implementations of term rewriting. As far as we know it is the first such calculus which is optimal in the sense of Lévy. Moreover, as far as we know this is the first optimal calculus featuring only a single scope delimiter node instead of the usual two, croissants and brackets, which by force eliminates the problems which are caused by having more than one scope node [2, Chapter 9]. The calculus is simple, half a page suffices (see Section 4) to describe it, and completely reduction-based (no semantic read-back in the implementation). As a consequence it can be trivially implemented in any (modern) programming language.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[3] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 2nd revised edition, 1984.

[4] R. Bird and R. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[5] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.

[6] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *LICS 14*, pages 193–202. IEEE Computer Society Press, 1999.

[7] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *POPL 19*, pages 15–26. ACM Press, 1992.

[8] D. Hendriks and V. v. Oostrom. ⅄. In *CADE 19*, volume 2741 of *LNAI*, pages 136–150. Springer, 2003.

[9] Y. Lafont. Interaction nets. In *POPL 17*, pages 95–108. ACM Press, 1990.

[10] J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL 17*, pages 16–30. ACM Press, 1990.

[11] J.-J. Lévy. *Réductions correctes et optimales dans le λ-calcul*. Thèse de doctorat d'état, Université Paris VII, 1978.

[12] K.-J. v. d. Looij. Tba. Master's thesis, Universiteit Utrecht, 2004. Forthcoming.

[13] I. Mackie. Efficient lambda evaluation with interaction nets. In *RTA 15*, 2004. Forthcoming.

[14] V. v. Oostrom. Net-calculus. Lecture notes for the course "Computationele Methoden in de Typen theorie" (in Dutch), 2001.

[15] M. Zwitserlood. End-of-scope, locally. Master's thesis, Universiteit Utrecht, 2004. Forthcoming.